Fysikkrelaterte oppgaver i programmering
IN1900

Juni 2017

With Python, life is much simpler.

Morten Hjorth-Jensen

Fysisk institutt

Forord

Disse oppgavene ble laget som et supplement til introduksjonskurset IN1900 som et initiativ fra Fysisk institutt. Ideen har vært å lage en samling av fysikkrelaterteoppgaver som svarer til vanskelighetsgraden til de oppgavene som har originalt vært brukt i emnet. Vi har også prøvd å skrive oppgavene slik at det skal ikke være nødvendig å kunne noe om fysikk for å løse noen av dem.

Hvis du finner noe feil ved oppgavene eller har andre spørsmål, kan du sende dem til

Jonas Gahr Sturtzel Lunde `jonassl@student.matnat.uio.no`
Kristine Baluka Hein `krisbhei@student.matnat.uio.no`

Innhold

Oppgave 1.1 - Massetetthet

Ulike materiale har ulike massetettheter. Massetettheten er definert masse delt på volum, oftest i $\mathrm{kg/m^3}$.

Materialer	Polystyren (lav tetthet)	Kork	Rhenium	Platinium
Massetetthet (i $\mathrm{kg\,/m^3}$)	20	220	21020	21450

Tabell 1: Massetettheten til ulike materialer

En kube veier 858 g og har volum $40\,\mathrm{cm^3}$. Skriv et program som finner massetettheten til kuben og bruk resultatet til å finne hvilket materiale kuben består av. Kuben består av et av materialene i tabellen over. Det kan hende at svaret ditt vil ikke være den eksakt en av de gitte massetetthetene siden massetetthet er i seg selv et gjennomsnittlig mål. Du kan velge materialet som har massetett som er nærmest ditt resultat.

Filnavn: `massdensity.py`

Oppgave 1.2 - Beregne solmassen

Det er mulig å regne ut solens masse ved å bruke at

$$M_{sol} = \frac{4\pi^2 \cdot (1\,\mathrm{AU})^3}{G \cdot (1\,\mathrm{yr})^2} \tag{1}$$

I denne oppgaven kommer vi til å bruke omtrentlige verdier for AU og G.
Enheten AU er en astronomisk lengdeenhet gitt ved avstanden mellom solen og jorden. Den har en verdi

$$1\,\mathrm{AU} = 1.58 \times 10^{-5}\,\mathrm{lysår}$$

der $1\,\mathrm{lysår} = 9.5 \times 10^{12}\,\mathrm{km}$.

Konstanten G kalles gravitasjonskonstanten og har verdi

$$G = 6.674 \times 10^{-11}\,\mathrm{m^3\,kg^{-1}\,s^{-1}}$$

Lag et program som regner ut solmassen ved å bruke ligning (1) og som skriver ut resultatet i kilogram med en pent formatert `print`. Resultatet ditt skal bli omtrent $M_{sol} \approx 2.01 \cdot 10^{30}\,\mathrm{kg}$.

Filnavn: `solarmass.py`

Oppgave 1.3 - Halveringstid

Et radioaktivt materie er et ustabilt stoff som over tid vil reduseres til andre materier mens det avgir radioaktiv stråling. Av en opprinnelig masse N_0 av et radioaktivt materie, vil den gjenværende massen etter en tid t være gitt ved

$$N(t) = N_0 \mathrm{e}^{-t/\tau} \tag{2}$$

τ er den såkalte 'tidskonstanten' til det radioaktive materiet, og representerer den gjennomsnittlige tiden et enkelt atom bruker før det nedbrytes. Den kan variere fra under $10^{-20}\,\mathrm{s}$ for ekstremt ustabile isotoper, til over $10^{30}\,\mathrm{s}$ for stoffer som for alle praktiske formål kan regnes som stabile.

a)

Karbon-11 er et ustabilt karbonisotop, og har en tidskonstant $\tau = 1760$ s.

Lag et program som beregner hvor mye som gjenstår av karbonet etter 10 minutter. Anta at vi starter med en mengde $N_0 = 4.5\,\text{kg}$ av karbon-11.

Hint: For å teste at programmet ditt fungerer som det skal, kan du sette t lik 0, og et meget stort tall, og sjekke at resultatene virker i overensstemmelse med slik du forventer et radioaktiv materiale å oppføre seg.

b)

Selv om tidskonstanten gir oss en veldig pen formel, snakker vi oftere om 'halveringstiden' $t_{\frac{1}{2}}$ til materiet, som er tiden det tar for halvparten av materiet å nedbrytes. Forholdet mellom tidskonstanten og halveringstiden er gitt ved

$$\tau = \frac{t_{\frac{1}{2}}}{\ln 2}$$

Halveringstiden til karbon-11 er $t_{\frac{1}{2}} = 1220$ s. Skriv om programmet ditt slik at det først regner ut tidskonstanten fra halveringstiden, og deretter beregner mengden av stoffet, som i forrige oppgave. Kontroller at du får samme svar som i forrige oppgave dersom du bruker samme masse og tid.

Filnavn: `half_life.py`

Oppgave 1.4 - Farten til et atom

Atomene i et materiale er strukturert slik at de danner et repeterende gitter mønster av deres plasseringer. Vi skal se på et atom som beveger seg langs overflaten til materialet. På grunn av gitterstrukturen, kan vi lage en modell der farten til atomet er periodisk:

$$v(x) = \sqrt{v_0^2 + \frac{2F_0}{m}\left(\cos\left(\frac{x}{n}\right) - 1\right)}$$

der m er atomets masse, x er posisjonen til atomet, v_0 er atomets startfart og n er en skalert avstand mellom atomene i materialet. Vi setter kraften $F_0 = 1$.
Finn farten til atomet når $x = 1$, $v_0 = 2$, $n = 4$ og $m = 3$.

Filnavn: `velocity_of_atom.py`

Oppgave 1.5 - Finn feil hos Einstein

Spesiell relativitetsteori er et område i fysikken som omhandler veldig store hastigheter. I spesiell relativitetsteori er bevegelsesmengden p til et objekt med masse m (i kg) og hastighet v (i m/s) gitt ved

$$p = m \cdot v \cdot \gamma, \qquad \gamma = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}$$

der $c \approx 300\,000\,000\,\text{m/s}$ er lyshastigheten. Programmet under forsøker å regne ut bevegelsesmengden til et objekt med en hastighet på $1/3$ av lyshastigheten, med masse $m = 0.14$ kg. Programmet er fullt av feil. Kopier programmet og kjør det. Rett opp feilene, og få programmet til å printe en korrekt p.

```
1 from math import squareroot
2 c = 300 000 000  # m/s
3 v = 100 000 000  # m/s
4 m = 0,14  # kg
5
6 gamma = 1/squareroot(1-(v^2/c^2))
7 p = m*v*gamma
8
9 print p
```

Filnavn: `Einsteins_errors.py`

Oppgave 1.6 - Rydbergs konstant

Rydbergkonstanten R_∞ for tunge atomer brukes i fysikken for å kunne beregne bølgelengden til spektrallinjer[1].

Konstanten er funnet til å være

$$R_\infty = \frac{m_e e^4}{8\varepsilon_0^2 h^3 c}$$

der

- $m_e = 9.109 \times 10^{-31}\,\mathrm{m}$ er massen til elektronet
- $e = 1.602 \times 10^{-19}\,\mathrm{C}$ er ladningen til et proton (også kalt for *elementærladningen*)
- $\varepsilon_0 = 8.854 \times 10^{-12}\,\mathrm{C\,V^{-1}\,m^{-1}}$ er den elektriske konstanten
- $h = 6.626 \times 10^{-34}\,\mathrm{J\,s}$ er Plancks konstant
- $c = 3 \times 10^8\,\mathrm{m/s}$ er lysets fart

Dette er fysiske konstanter som brukes en god del i fysikk, og som du høyst sannsynlig kommer til å møte på flere ganger i andre fysikkemner!

Lag et program som setter de fysiske konstantene som variabler og bruker variablene til å regne ut verdien til Rydbergs konstant. Verdien til Rydbergskonstant skal da skrives til skjerm ved hjelp av `print`.

Se om programmet ditt gir at

$$R_\infty = 10961656.2162 \quad (\mathrm{i\ m^{-1}})$$

Merk at dette er kun en omtrentlig verdi. Dette er fordi vi har rundet av verdiene de fysiske konstantene til tre desimaler.

Filnavn: `constant_Rydberg.py`

[1] Årsaken til at vi bruker ∞ i symbolet til konstanten er fordi vi antar at massen til atomets kjerne er uendelig stor sammenlignet med massen til elektronet.

Innhold

Oppgave 2.1 - Måle tid

En ball slippes ned fra et stup i et rettlinjet bevegelse ved høyde h. Posisjonen til ballen etter en tid t kan uttrykkes ved

$$y(t) = v_0 t - \frac{1}{2}at^2 + h$$

der a er akselerasjonen i $\mathrm{m/s^2}$, h er høyden (i meter) der ballen slippes fra og v_0 er startfarten (målt i m/s) .

Vi ønsker å finne ut ved hvilken tid t_1 ballen passerer en spesifikk høyde y_1. Vi ønsker altså å finne verdien til t_1 slik at $y(t_1) = y_1$. Ballens posisjon blir målt per Δt sekund.

Lag et program som finner ut hvor lang tid det tar (t_1) før ballen passerer høyden y_1. Programmet skal bruke en while loop.

Her lar vi $h = 10\,\mathrm{m}$, $y_1 = 5\,\mathrm{m}$, $\Delta t = 0.01\,\mathrm{s}$, $v_0 = 0\,\mathrm{m/s}$ og $a = 9.81\,\mathrm{m/s^2}$.

Hint: Vi kan ikke bruke '`==`' i while loopen for å finne tiden t_1 når ballen er ved høyden y_1. Dette er fordi vi måler tiden pr. Δt sekunder. Vi vil sannsynligvis måle posisjonen til ballen *etter* den har nådd y_1. Det beste vi kan gjøre, er å øke tiden med Δt så lenge høyden til ballen er *større* enn y_1. Med en gang ballen har en høyde som er mindre eller lik y_1 ved en tid, så kan vi sette denne tiden til å være t_1. Programmet vårt vil derfor være noe unøyaktig, men dette er det beste vi kan få til siden vi ikke kan få uendelig liten Δt.

Filnavn: `throw_ball_height.py`

Oppgave 2.2 - Relativistisk bevegelsesmengde

Fra klassisk fysikk har vi at bevegelsesmengden p til et objekt med masse m og hastighet v er gitt ved

$$p = m \cdot v$$

En satellitt med masse $1200\,\mathrm{kg}$ er fanget i nærheten av et sort hull, og akselererer raskt fra $v = 0$ til $v = 0.9c$, der c er lyshastigheten i vakuum, som er $c \approx 3 \times 10^8\,\mathrm{m/s}$.

a)

Skriv et program som printer to pene kolonner til terminalen: Satellittens hastighet i intervaller på $0.1c$, og satellittens bevegelsesmengde ved disse hastighetene.

Hint: Bruk vitenskapelig notasjon, '%e', når du skriver verdiene, for å slippe ekstremt store floats. Eventuelt '%g', som velger notasjon for deg. Prøv å ikke få med unødvendig mange desimaler i printen.

b)

I oppgave 1.5 så vi at bevegelsesmengde i spesiell relativitetsteori er definert som

$$p_{rel} = m \cdot v \cdot \gamma, \qquad \gamma = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}$$

Dette er den faktiske bevegelsesmengden til et objekt, men vi skal nå se at den klassiske formelen er en veldig god tilnærming ved 'lave' hastigheter.

Utvid programmet ditt fra oppgave a, slik at det printer tre kolonner: Hastigheten, den klassiske bevegelsesmengden, og den relativistiske bevegelsesmengden.

Filnavn: `relativistic_momentum.py`

Oppgave 2.3 - Radioaktiv liste

a)

Ta utgangspunkt i formelen for radioaktiv nedbrytning fra oppgave 1.2:

$$N(t) = N_0 \mathrm{e}^{-t/\tau}$$

Lag en while loop som fyller to lister: En med tidspunkter t, og en med verdier av $N(t)$ ved disse tidspunktene. Start i $t = 0$ og bruk tidssteg på 60 s. Loopen skal avbrytes når den gjenværende massen av stoffet er under 50% av den opprinnelige. Vi ser fortsatt på en masse $N_0 = 4.5\,\mathrm{kg}$ av karbon-11, med tidskonstant $\tau = 1760\,\mathrm{s}$.

Fra oppgave 1.3 husker vi at *halveringstiden* til et materie er den tiden det tar for halvparten av materiet å nedbrytes. Fordi vi avbrøt loopen når halvparten av materiet var igjen, bør det siste elementet i tidslisten din være halveringstiden $t_{\frac{1}{2}}$ til karbon-11.

Sjekk at dette stemmer ved å sammenligne den siste N-verdien i listen din med halveringstiden som definert i oppgave 1.3. Husk at verdien du målte kan avvike på opptil et minutt, fordi vi brukte tidssteg på et minutt mellom hver måling.

b)

Kombiner listene til en nøstet liste, `Nt`, slik at hvert element i listen `Nt` er et par av tilhørende t- og $N(t)$-verdier. For eksempel skal det første elementet `Nt[0]` være en liste av `[0, 4.5]`. Bruk den nye nøstede listen til å skrive ut et pent table av alle de tilhørende t og $N(t)$ verdier til terminalen.

Filnavn: `radioactive_list.py`

Oppgave 2.4 - Newtons universelle gravitasjonslov

Newtons gravitasjonslov beskriver hvordan gravitasjonen som en tiltrekningskraft virker mellom to legemer:

$$F = G\frac{m_1 m_2}{r^2}$$

der m_1 og m_2 er massene til de to legemene som tiltrekker hverandre og r er avstanden mellom dem.

Konstanten G er gravitasjonskonstanten som har følgende verdi:

$$G = 6.674 \times 10^{-11}\,\mathrm{m^3\,kg^{-2}\,s^{-1}}$$

Vi ser på et legeme med masse $M = 3\,\mathrm{kg}$ som blir påvirket av N legemer med masse $m_1, m_2, \ldots, m_N$. Det i-te legeme har masse $m_i = \frac{i}{6} + 2\,\mathrm{kg}$, og har avstand $r_i = \sqrt{\left(\frac{i}{4}\right)^2 + 10}$ meter fra legemet med masse M.

Antall legemer er $N = 10$.

Skriv et program som beregner den totale kraften $\sum_{i=1}^{N} F_i = F_1 + F_2 + ... + F_N$ som virker på legemet med masse M.

Filnavn: `newton_gravitation.py`

Oppgave 2.5 - Fanget kvantepartikkel

Kvantemekanikk er et område i fysikken som omhandler virkeligheten på veldig små skalaer. En av resultatene i kvantemekanikk er at partikler noen ganger bare har lov til å ha spesifikke energinivåer, og kan aldri har energier mellom disse nivåene. Partiklene må da hoppe fra ett energinivå til et annet.

Kvantemekanikk sier at en partikkel som er fanget i en veldig liten 'boks'[2] av lengde L, har det bare lov til å ha energiene:

$$E_n = \frac{n^2 h^2}{8mL^2}, \qquad n = 1, 2, 3 \ldots$$

hvor m er partikkelens masse, og h er plancks konstant, $h = 6.626 \times 10^{-34}\,\mathrm{J\,s}$.

Vi skal se på et elektron med masse $9.11 \times 10^{-31}\,\mathrm{kg}$ som fanget i en boks av lengde $10^{-11}\,\mathrm{m}$. Elektronet starter på det laveste energinivået, E_1 (IKKE E_0!), og hopper så oppover, ett energinivå om gangen. Hvert hopp fra et energinivå E_i til E_{i+1} vil kreve en energi

$$E_{i+1} - E_i = \frac{((i+1)^2 - i^2)h^2}{8mL^2}$$

Skriv en for loop som beregner energien er nødvendig for hver steg, og lagrer hver verdi i en liste. Summer også opp den totale energien som kreves gjennom alle hoppene.

Filnavn: `quantum_trap.py`

Oppgave 2.6 - Loope over radiuser til looper

For at en skal kunne kjøre gjennom en loop, er det nødvendig å ha en fart som er større enn en viss grense.

Anta en stuntperson skal kjøre gjennom en loop. For at stuntpersonen skal være i kontakt med loopen gjennom hele stuntet, må farten v v*minst* være:

$$v = \sqrt{gr}$$

Her er $g = 9.81\,\mathrm{m/s^2}$ og r er radiusen til loopen (i meter).

Du har blitt gitt et program som bruker en while loop for å iterere gjennom en liste av radiuser r til ulike looper. For hver radius r programmet iterere gjennom blir den minste farten v beregnet og deretter vist til skjerm:

```
from math import sqrt

g = 9.81
r = [2.7,3.43,5.62,7.1]
num_loops = len(r)
v = [0]*num_loops

i = 0
while i < num_loops:
```

[2] Et uendelig dypt kvadratisk potensial

```
    v[i] = sqrt(r[i]*g)      #in m/s
    v[i] = v[i]*3600/1000    #convert to km/h
    print "Least speed to complete the loop: %.2f km/h"%v[i]
    i += 1
```

Programmet heter `while_loop_over_loops.py` som du kan finne her: (lenke til program).

Skriv om programmet slik at den bruker en for loop isteden. Endre også programmet slik at den ikke printer v for hver radius i loopen, men heller printer verdiene i en egen for loop etter alle v har blitt regnet ut.

Filnavn: `for_loop_over_loops.py`

Oppgave 3.1 - Radioaktiv funksjon

Implementer formelen for radioaktiv nedbrytning fra oppgave 1.3 som en funksjon `N(N0, tau, t)`.

Skriv en test-funksjon som bruker parametrene fra oppgave 1.3a, $N_0 = 4.5\,\text{kg}$, $\tau = 1760\,\text{s}$, $t = 600\,\text{s}$, og sjekk gjerne at svaret blir $3.2\,\text{kg}$ (som var resultatet fra 1.3a) innenfor en toleranse.

Ikke sett toleransen mindre enn 10^{-4}, ettersom 3.2 ikke er helt nøyaktig.

Filnavn: `radioactive_function.py`

Oppgave 3.2 - Kvantefunksjon

I oppgave 2.5 så vi at partikler fanget i en boks på størrelse L bare kan ha energier

$$E_n = \frac{n^2 h^2}{8mL^2}, \qquad n = 1, 2, 3 \ldots$$

hvor m er partikkelens masse, og h er Planck's constant: $h = 6.626 \times 10^{-34}\,\text{J s}$

Skriv en funksjon `quantum_energy`, som tar to energinivåer[3], lengden på boksen og partikkelens masse som argumenter, og returnerer energidifferansen mellom de to nivåene.

Filnavn: `quantum_function.py`

Oppgave 3.3 - Trekloss

I denne oppgaven skal du lage et program som finner ut hvor langt en trekloss med startfart $v_0\,\text{m/s}$ vil komme når den sklir over ulike underlag bestående av ulike materialer. Materialet underlaget består av vil påvirke hvilken friksjonskraft som virker på klossen.

Posisjonen til klossen etter en tid t kan uttrykkes som:

$$x(t) = v_0 t - \frac{1}{2}\mu g t^2$$

der μ er en friksjonskoeffisient og $g = 9.81\,\text{m/s}^2$. Vi kan regne ut ved hvilken tid T klossen stopper. Tiden T er funnet til å være

$$T = \frac{v_0}{\mu g}$$

Definer en funksjon som tar inn en liste av ulike friksjonskoeffisienter, beregner $x(T)$ og lagrer hvert resultat i en liste. Listen skal så returneres.

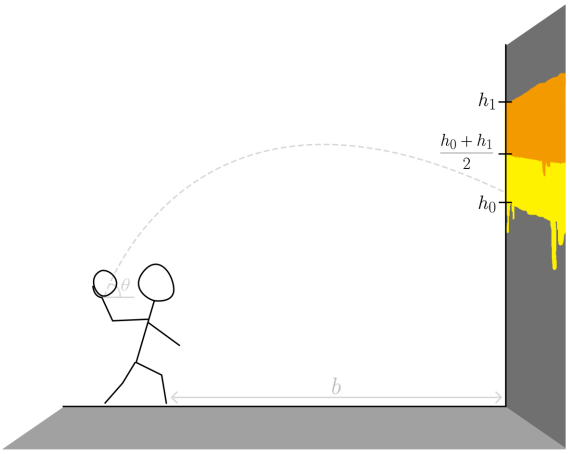
La $v_0 = 5\,\text{m/s}$ og listen over friksjonskoeffisienter være `[0.62,0.3,0.45,0.2]`. Gjør så et kall på funksjonen og skriv ut resultatene fra funksjonen sammen med tilhørende friksjonskoeffisienter.

Filnavn: `block_frictions.py`

Oppgave 3.4 - Treffe blink

Vi skal se på hvordan vi kan, ved hjelp av en enkel modell, simulere et spill. Dette spillet går ut på at en person skal treffe en blink malt på en vegg med en ball. Personen får poeng avhengig av hvor på veggen ballen treffer.

[3] Funksjonen skal ta inn indeksen på de to energinivåene, n.



Figur 1: Illustrasjon av systemet vi skal basere våres spillsimulering på.

Ballens høyde over bakken kan modelleres ved

$$y(t) = -\frac{1}{2}gt^2 + v_0 t \sin\theta$$

der v_0 er farten personen kaster ballen med, θ er kastevinkelen og $g = 9.81\,\mathrm{m/s^2}$.

a)

Skriv en funksjon som returnerer høyden til ballen ved en tid t.

b)

Fra modellen kan vi regne oss fram til at ballen vil treffe veggen ved tiden $T = \dfrac{b}{v_0 \cos\theta}$, der b er avstanden mellom personen og veggen.

Vi må se på verdien av $y(T)$ for å finne ut hvor mange poeng personen skal få. Antall poeng skal beregnes og returneres fra en funksjon du skal skrive selv.

Blinken er malt slik at den dekker veggen mellom høyden h_0 og høyden h_1 der $h_0 < h_1$. Poengene gis på følgende måte:

- Personen får 0 poeng dersom $y(T) < h_0$ eller $y(T) > h_1$
- Personen får 1 poeng dersom $h_0 \leq y(T) < \frac{1}{2}(h_1 + h_0)$
- Personen får 2 poeng dersom $\frac{1}{2}(h_1 + h_0) \leq y(T) \leq h_1$

Skriv et program som skriver ut i en for loop hvor mange poeng personen får ved å kalle på din nyskrevne funksjon dersom $h_0 = 3\,\mathrm{m}$, $h_1 = 3.5\,\mathrm{m}$, $\theta = \dfrac{\pi}{4}$, $b = 3.5\,\mathrm{m}$ for $v_0 = 15, 16, 19, 22\,\mathrm{m/s}$.

Filnavn: `hit_target.py`

Oppgave 3.5 - Kast ned stup

a)

Vi kaster en ball ned et stup, i en rettlinjet bevegelse med konstant akselerasjon. Vi ignorerer luftmotstand, og regner bare i én dimensjon. Hastigheten til ballen kan beregnes enten som en funksjon av distansen ballen har falt, x, eller tiden fra vi kastet den, t.

$$v = v_0 + at \tag{3}$$

$$v = \sqrt{v_0^2 + 2ax} \tag{4}$$

(Den nederste er kanskje mer kjent på formen $v^2 - v_0^2 = 2ax$).

v_0 er initialhastigheten ved $t = 0$, og a er akselerasjonen, som på jordoverflaten er $a = 9.81\,\mathrm{m/s^2}$. Vi regner positiv retning nedover, slik at posisjon, initialhastigheten og akselerasjonen alle er i positiv retning.

Skriv to funksjoner basert på ligningene over, `velocity1` og `velocity2`, som returnerer hastigheten fra de tre variablene.

Filnavn: `cliff_throw.py`

b)

Skriv en testfunksjon `test_velocity()`, som sjekker om begge hastighetsfunksjonene gir samme hastighet ved et gitt tidspunkt, med samme initialhastighet. Du kan regne distansen ballen har falt ved et gitt tidspunkt fra formelen

$$x = v_0 t + \frac{1}{2} a t^2$$

c)

Utvid `velocity1` fra oppgave a slik at t også kan være en liste av tidspunkter, og funksjonen skal i så fall returnere en liste av hastigheter. Merk at t fortsatt også skal kunne være et tall, og funksjonen skal i så fall returnere en enkelt hastighet, akkurat som før.

Hint: Bruk en if-else block til å sjekke typen til t, og så en for loop som fyller en liste av hastighetsverdier. Husk at 'et tall' kan være enten en float eller en integer.

Oppgave 3.6 - Enda en trekloss

Vi ønsker å lage et program som finner ut hvor langt en trekloss med startfart v_0 vil komme når den sklir over underlag som består av ulike materialer. Materialet underlaget består av, vil påvirke hvilken friksjonskraft som vil virke på klossen.

Farten og avstanden til klossen ved en tid t kan uttykkes ved:

$$v(t) = v_0 - \mu g t$$

$$x(t) = v_0 t - \frac{1}{2} \mu g t^2$$

der μ er en friksjonskoeffisient og v_0 er farten til klossen ved starttiden $t = 0$.

a)

Definér en funksjon som returnerer avstanden $x(t)$ og som tar inn t, v_0 og en friksjonskoeffisient μ som parametre.

b)

Definér en funksjon som returnerer en liste over hvor langt klossen skled for hver av de ulike friksjonskoeffisienten.

La funksjonen ta inn Δt som parameter som er hvor store tidssteg vi ønsker å gjøre pr. tidsmåling av farten til klossen.

Funksjonen skal så bruke en while loop for å finne tiden t når farten $v(t)$ blir mindre eller lik 0. Bruk tiden t som har blitt funnet til å beregne avstanden $x(t)$. Avstanden skal da lagres i en liste. Listen skal så returneres når alle avstandene er ferdig beregnet.

Sett $v_0 = 5\,\mathrm{m/s}$, $\Delta t = 0.0001\,\mathrm{s}$ og listen over friksjonskoeffisienter til å være `[0.62,0.3,0.45,0.2]`.

c)

Skriv en testfunksjon som tester om avstandene fra b) er lik den analytiske avstanden $x\left(\frac{v_0}{\mu g}\right)$ (som er den analytiske tiden hver kloss vil bruke) for hver friksjonskoeffisient μ. En passende grense for å teste om de beregnede avstandene er lik den analytiske avstanden er 10^{-7} i denne oppgaven.

Kall tilslutt på testfunksjonen for å sjekke om din implementasjon fra b) stemmer overens med den analytiske avstanden for hver friksjonskoeffisient μ.

Filnavn: `block_frictions2.py`

Oppgave 4.1 - Heisenbergs uskarphetsrelasjon

Heisenberg viste i 1927 at vi ikke kan vite eksakt en partikkels fart og posisjon *samtidig*. Dette betyr at dersom vi vet ganske så nøyaktig hva posisjonen til en partikkel er, vil vi ikke vi ikke vite like nøyaktig hva farten til partikkelen er, og vice versa.
Matematisk kan en skrive dette slik

$$\Delta x \Delta p \geq \frac{h}{4\pi}$$

der Δx er usikkerhet (et mål på hvor nøyaktig målingen er) av partikkelens posisjon og Δp er usikkerheten av partikkelens bevegelsesmengde[4].

Vi bruker at $h \approx 6.626 \times 10^{-34}\,\mathrm{J\,s}$.

a)

Skriv et program som tar inn Δx og Δp som argumenter på kommandolinjen. Programmet skal så sjekke argumentene i en `try-except` block og gi feilmelding dersom det ikke har blitt gitt nok argumenter og argumentene kan ikke konverteres til flyttall.

b)

Definér en funksjon der Δx og Δp sendes inn som parameter. Testfunksjonen skal sjekke om uskarphetsrelasjonen holder for de gitte Δx og Δp. Hvis relasjonen ikke er oppfylt, skal en passende feilmelding skrives. La funksjonen bruke `assert` til å teste relasjonen.

Test programmet ditt for tilfellene der

$\Delta x_1 = 3.101\,65 \times 10^{-9}\,\mathrm{m}$, $\Delta p_1 = 1.7 \times 10^{-26}\,\mathrm{kgm\,/s}$

og

$\Delta x_2 = 5.2 \times 10^{-32}\,\mathrm{m}$, $\Delta p_2 = 1 \times 10^{-3}\,\mathrm{kgm\,/s}$.

Usikkerhetene Δx_1 og Δp_1 bryter ikke med uskarphetsrelasjonen. Derimot usikkerhetene Δx_2 og Δp_2 gjør det (og derfor skal programmet ditt gi feilmelding for dette tilfellet).

Hint: For å representere en ganske så lav verdi som f.eks 1.7×10^{-26} på kommandolinjen, kan du skrive `1.7e-26`.

Filnavn: `uncertainty_Heisenberg.py`

Oppgave 4.2 - Partikkelakselerator

Et elektrisk felt med feltstyrke E vil påvirke en partikkel med ladning q med en kraft $F = qE$.
En partikkel med initialhastighet v_0 etter en tid t ha en posisjon og hastighet gitt som

$$x(t) = v_0 t + 0.5\frac{qE}{m}t^2$$

and

$$v(t) = v_0 + \frac{qE}{m}t$$

[4]Bevegelsesmengden er definert ved farten til partikkelen. Dersom vi vet bevegelsesmengden og massen til partikkelen, vil vi også vite dets fart. Bevegelsesmengden p er definert som $p = mv$ der m er massen til legemet vi ser på (i vårt tilfelle: en partikkel) og v farten til legemet vi ser på.

a)

Elektroner har masse $m \approx 9.1 \times 10^{-31}\,\text{kg}$ og elektrisk ladning $q \approx -1.6 \times 10^{-19}\,\text{C}$. Vi skal se på et elektron fanget i et elektrisk felt av styrke $E = 0.02\,\text{N/C}$. [5]

Skriv et program som spør brukeren etter verdier for v_0 og t, og printer posisjonen og hastigheten til elektronet ved dette tidspunktet.

Test programmet ditt ved tidspunktet $t = 15\,\text{s}$ og initialhastigheten $v_0 = 220\,\text{m/s}$.

b)

Skriv om programmet ditt slik at v_0 og t, samt nå også q og m, hentes fra terminalen.
Bruk en try/except blokk til å initialisere variablene, i tilfelle brukeren gir for få argumenter, eller de ikke kan konverteres til floats. I så fall skal programmet ditt spørre brukeren etter parametrene slik som i deloppgave a).

Protoner har masse $m \approx 1.67 \times 10^{-27}\,\text{kg}$ og elektrisk ladning $q \approx 1.6 \times 10^{-19}\,\text{C}$. Nøytroner har (tilnærmet) lik masse som protoner, og ingen ladning.

Print posisjonene og hastigheten til disse to partikklene med de samme parametrene som i deloppgave a).

Filnavn: `particle_accelerator.py`

Oppgave 4.3 - Relativistisk brukerinput

I oppgave 2.2 sammenlignet vi den relativistiske og klassiske bevegelsesmengden til et objekt med masse m og hastighet v.

$$p_{clas} = m \cdot v$$
$$p_{rel} = m \cdot v \cdot \gamma, \qquad \gamma = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}$$

a)

Skriv et program som spør om hastighet og masse, og regner ut objektets bevegelsesmengde. Skriv programmet ditt slik at det bruker den klassiske formelen dersom den gitte hastigheten er såpass lav at den klassiske er en god tilnærming, og den relativistiske ellers. Bruk resultatene dine fra oppgave 2.2 til å bestemme ved hvilke hastigheter den klassiske formelen er en grei nok tilnærming.

Dersom du ikke har gjort 2.2 kan du bruke den klassiske for hastigheter under $v = 1/3c \approx 10^8\text{m/s}$.

b)

Skriv om programmet ditt slik at det henter m og v fra terminalen istedenfor som keyboard input.

[5] Vi ser bare på bevegelse i én dimensjon. Merk at et elektron, på grunn av sin negative ladning, vil påvirkes av en negativ akselerasjon, med mindre feltstyrken også er negativ.

c)

Utvid programmet ditt fra oppgave b) til å grundig teste input'ene den får ved å implementere følgende tester:

- En `Try/Except` blokk til å initiere variablene. Du må inkludere både en `ValueError` og en `IndexError` i `Except`'en din, dersom det enten gis for få terminal argumenter, eller de ikke kan konverteres til float.
- En test med en `ValueError` dersom den gitte massen ikke er positiv.
- En test med en `ValueError` dersom *absoluttverdien* til hastigheten er større en lyshastigheten.

Få med en forklarende feilmelding til alle testene.

Filnavn: `momentum_input.py`

Oppgave 4.4 - Hvor stor friksjon?

Anta at du har et utvalg bøker plassert på en helning med vinkel θ.

For at en bok skal begynne å skli, må friksjonskraften f være

$$f = \mu_s mg \cos\theta$$

der m er bokens masse i kg, μ_s den statiske friksjonskoeffisienten og $g = 9.81\,\mathrm{m/s^2}$. Den statiske friksjonskoeffisienten kan variere ettersom hvilket materiale helningen er laget av.

Vi har fått et samling av data, `slide_books.dat`, over bøkene. Filen inneholder informasjon on bøkenes ulike masse m, ulike vinkler θ og statiske friksjonskoeffsienter μ_s.

a)

Lag et program som åpner `slide_books.dat` og leser inn de ulike verdiene for m,θ og μ_s. Programmet ditt skal skrives slik at den skal kunne lese inn et vilkårlig antall verdier for m, θ og μ_s.

b)

Utvid programmet ditt fra a) og bruk de innleste verdiene til å beregne hvor stor friksjonen må være for at hver bok per helningsvinkel og friksjonskoeffisient skal begynne å skli.

Pass på at vinklene i `slide_books.dat` er gitt i grader når programmet ditt regner ut friksjonskraften f! For å konvertere gradene til radianer, kan du bruke at

$$\text{vinkel i radianer} = \frac{(\text{vinkel i grader}) \cdot \pi}{180}$$

Resultatene skal tilslutt skrives til fil. Formatet må være noe tilsvarende som dette eksempelet:

```
--- Book with mass 4.33 kg ---
theta = 0.93 rad
coefficient of friction = 0.34
needed friction force is 8.69 N

coefficient of friction = 0.2
needed friction force is 5.11 N
```

```
coefficient of friction = 0.55
needed friction force is 14.06 N

coefficient of friction = 0.4
needed friction force is 10.23 N
```

Filnavn: `slide_books_friction.py`

Oppgave 4.5 - Newtons gravitasjons lov

I oppgave 2.4 så vi på hvordan gravitasjonskraften mellom to legemer virker mellom seg ved å bruke Newtons gravitasjonslov:

$$F = G\frac{m_1 m_2}{r^2}$$

der m_1 og m_2 er massene til legemene og r er avstanden mellom dem.

Konstanten G er gravitasjonskonstanten som har verdi

$$G = 6.674 \times 10^{-11}\,\mathrm{m^3\,kg^{-2}\,s^{-1}}$$

Vi skal se på hvor mye gravitasjonskraften virker mellom et legeme med masse M kg og N andre legemer. Det i-te legemet har masse m_i kg og avstand r_i fra legemet med masse M.

Skriv et program som leser inn en fil og bruker verdiene i filen til å beregne og skrive ut den totale gravitasjonskraften som virker på legemet med masse M.

Du kan anta at filene som programmet ditt skal lese, har følgende struktur:

```
0.2     0.0034
3       0.495e-5
0.15    2
2.5     0.000029348
1.3948  4.56
4.5     0.0294
```

Filen heter `newton_objects.dat` som du kan finne her: (lenke til fil).

Den i-te linjen består av informasjon om det i-te legemet. Den første verdien er m_i og andre verdien er r_i.

Programmet skal ta inn massen M som første argument og filnavnet bestående av informasjon om de N legemene som andre argument på kommandolinjen.

Programmet skal så gjøre følgende i en `try-except`-block:

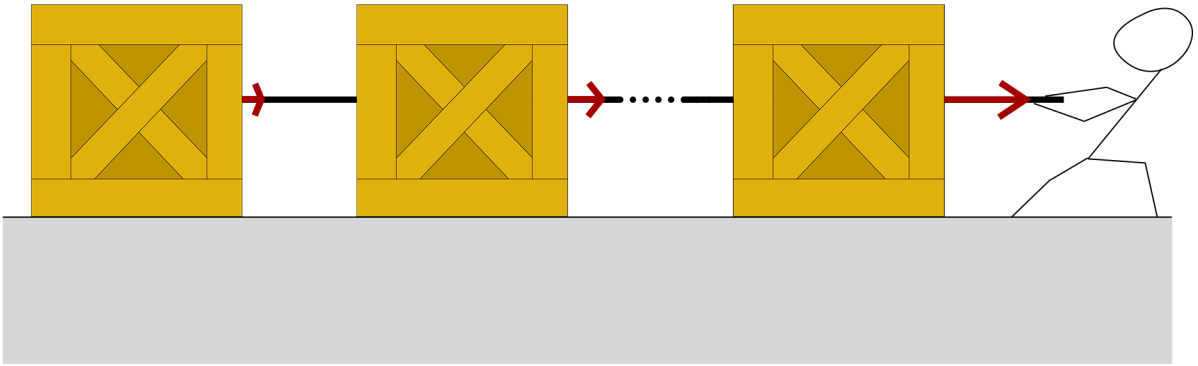
- Skrive en passende feilmelding og avslutte programmet dersom brukeren glemmer å skrive massen M eller filnavnet som argumenter på kommandolinjen.
  Med andre ord, skal feilmeldingen vises dersom det har oppstått en `IndexError`.
- Skrive en passende feilmelding og avslutte dersom brukeren skriver inn noe for massen M som ikke kan konverteres til flyttall. Da har en `ValueError` oppstått.
- Skrive en passende feilmelding og avslytte dersom den gitte filen ikke eksisterer. Da vil det oppstå en `IOError`.

La programmet ditt lese inn `newton_objects.dat` og sett $M = 0.7\,\mathrm{kg}$.

Filnavn: `newton_gravitation_file.py`

Oppgave 5.1 - Dra kasser

Vi skal se på tilfellet der N kasser som er festet sammen med et tau blir dratt av en person.



Figur 2: Illustrasjon av en person som drar noen kasser.Prikkene mellom kassene indikerer at det er vilkårlig mange kasser. De røde pilene indikerer retninger og styrken på kreftene som virker på kassene.

a)

Vi har målt at hver i-te kasse blir påvirket av en kraft på $30 + 5 \cdot i\,\text{N}$. Lag et program som generer en liste av kreftene som virker på hver i-te kasse. Her har vi $N = 10$ kasser.

b)

Det viser seg at de kreftene som ble målt i a), viste seg til å være for store. Del verdiene fra a) på 2 *uten* å bruke for- eller while loops. Summér så verdiene, også uten å bruke for- eller while loops, og skriv ut resultatet.

Hint: Her må vi bruke arrays.

Filnavn: `pull_blocks.py`

Oppgave 5.2 - Plotting av relativistisk og klassisk bevegelsesmengde

Bevegelsesmengden til et objekt med masse m og hastighet v er definert forskjellig i klassisk fysikk og i relativitetsteori.

$$p_{clas} = m \cdot v$$

$$p_{rel} = m \cdot v \cdot \gamma, \qquad \gamma = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}$$

Lyshastigheten er definert ved $c \approx 3 \cdot 10^8 \text{m/s}$.

Sett $m = 5\,\text{kg}$ og plott funksjonene i samme plott med hastigheter jevnt fordelt i intervallet $v \in [0c,\ 0.9c] = [0\,\text{m/s},\ 2.7 \times 10^8\,\text{m/s}]$.

Filnavn: `momentum_plot.py`

Oppgave 5.3 - Kondensatorutladning

Fysisk introduksjon: En kondensator består av to metallplater satt opp parallelt mot hverandre. Hvis vi lader opp hver plate med henholdsvis positiv og negativ ladning (f.eks. ved å koble de til hver ende i et batteri), vil det dannes en spenning V over kondensatoren. Hvis vi deretter kobler platene sammen med en ledning, vil strøm begynne å gå mellom platene, og kondensatoren vil utlades. For å hindre det i å gå uendelig mye strøm, setter vi på en motstand R på ledningen mellom platene. Vi har nå en RC-krets.

Ladningen Q til en kondensator som utlades i en RC krets er over tid gitt som

$$Q(t) = CV\mathrm{e}^{-t/RC}$$

Følgende program regner ut denne utladningen for $n = 1000$ tidssteg over et intervall $t = [0\,\mathrm{s},\ 10\,\mathrm{s}]$. Kondensatoren har en kapasitans $C = 0.007\,\mathrm{F}$, en initiell spenning $V_0 = 50\,\mathrm{V}$, og kretsen har en motstand $R = 350\,\Omega$.

```
import numpy as np
import matplotlib.pyplot as plt

def I(t, R, C, V0):
    return C*V0*np.exp(-t/(R*C))

V0 = 50.0
R = 350.0
C = 0.007

t = 10
n = 1000
dt = float(t)/n

t_list = []
I_list = []
for i in range(n):
    t_list.append(dt*i)
    I_list.append([I(t, R, C, V0))

plt.plot(t_list, I_list)
plt.show()
```

Kopier programmet og sjekk at det kommer opp et plott som synker i en bue.

Vektoriser programmet slik at `t_list` og `I_list` erstattes med numpy-arrays, og for loopene erstattes med vektor-operasjoner. Sjekk at resultatet fortsatt blir det samme.

Filnavn: `capacitor_vectorization.py`

Oppgave 5.4 - Plancks Lov

Plancks lov beskriver hvor mye energi et svart legeme (som regel en stjerne) utstråler over et spekter av bølgelengder. Loven er gitt ved

$$B(\lambda) = \frac{2hc^2}{\lambda^5}\frac{1}{\mathrm{e}^{\frac{hc}{\lambda kT}} - 1}$$

der T er temperaturen til stjernen, $h = 6.626 \times 10^{-34}\,\mathrm{J\,s}$ er Planck's konstant, og $k = 1.38 \times 10^{-23}\,\mathrm{J/K}$ er Boltzmanns konstant. Lysets hastighet er fortsatt gitt som $c \approx 3 \times 10^8\,\mathrm{m/s}$. Kurvene som fremkommer av å plotte denne funksjonen over et spekter av bølgelengder kalles 'Planck-kurver', og de er et vanlig syn i fysikkbøker.

a)

Sett temperaturen lik solens temperatur, $T = 5800\text{K}$, og plott $B(\lambda)$ med bølgelengder jevnt fordelt i intervallet $\lambda \in [10\,\text{nm},\ 3000\,\text{nm}] = [10^{-8}\,\text{m},\ 3 \times 10^{-6}\,\text{m}]$

b)

Inkluder også Planck-kurvene til stjernene Alpha Centauri A ($T = 25\,000\,\text{K}$) og Proxima Centauri ($T = 2400\,\text{K}$) i samme plott med Solen.

c)

Wien's forskyvningslov forteller oss at toppunktet til Planck-kurvene skal ligge på bølgelengden

$$\lambda_{max} = \frac{b}{T}$$

der $b = 2.9 \times 10^{-3}\,\text{Km}$ kalles Wien's forskyvningskonstant.

Utvid plottet fra oppgave b) til å inkludere tre vertikal linjer ved $x = \lambda_{max}$ for hver av stjernene, og se at det stemmer med toppunktene til kurvene.

Hint: Du kan lage vertikale linjer med funksjonen `matplotlib.pyplot.axvline(x= )`

Filnavn: `Planck_curves.py`

Oppgave 5.5 - Oscillerende fjær

Hvis du henger en stein i enden av en fjær, trekker den ned en lengde A og slipper, vil steinen oscillere opp og ned med en vertikal posisjon gitt etter formelen

$$y(t) = A \cdot \mathrm{e}^{-\gamma t} \cos\left(\sqrt{\frac{k}{m}} \cdot t\right)$$

Høyden $y = 0$ korresponderer til posisjonen steinen vil ha når den henger løst i fjæren.
Steinen har en masse $m = 9\,\text{kg}$, og du trekker den ned en lengde $A = 0.3\,\text{m}$. Sett $k = 4\,\text{kg/s}^2$ og $\gamma = 0.15\,\text{s}^{-1}$.

a)

Lag to tomme arrays `t_array` og `y_array` av lengde 101. Bruk en for loop til å fylle dem med tidsverdier i intervallet $[0\,\text{s},\ 25\,\text{s}]$, og korresponderende $y(t)$ verdier.

b)

Vektoriser koden ved bruk av numpy sin `linspace` funksjon, til å generere arrayet `t_array`, og send den inn i en funksjon `y(t)` for å generere verdiene i `y_array`. Programmet ditt skal på dette tidspunkt ikke inneholde noen loops.

c)

Plott posisjonen til steinen mot tid i det gitte tidsintervallet. Bruk arrayene fra både deloppgave a) og b), og sjekk at grafene ligger helt oppå hverandre (sannsynligvis vil du ikke se den ene). Få korrekte enheter på begge akser.

Filnavn: `oscilating_spring.py`

Oppgave 5.6 - Planetbaner

Vi kan beskrive bevegelsen til en planet rundt en stjerne ved dens avstand til sin stjerne som en funksjon av hvor i banen planeten er:

$$r(\theta) = \frac{p}{1 + e\cos(\theta)} \tag{5}$$

der r er avstanden fra stjernen, og θ er hvilken vinkel i banen planeten har kommet til. $\theta = 0$ representerer vinkelen der planeten er nærmest stjernen.

p er en parameter som sier noe om størrelsen på banen. Vi setter den til 1 AU. [6]

e kalles banens *eksentrisitet*, og forteller oss hvor eliptisk banen er. $e = 0$ representerer en helt sirkulær bane, mens $e \geq 1$ beskriver en planet som ikke engang er i bane rundt stjernen, men bare passerer forbi.

a)

Plott planetens distanse fra stjernen som en funksjon av dens angulære posisjon, θ, for $e = 0$, $e = 0.5$ og $e = 0.8$ i samme plot. Bruk vinkler i intervallet $\theta \in [0,\ 2\pi]$, som altså tilsvarer én full bane rundt stjernen (vi regner i radianer). Få med korrekte enheter på begge akser.

b)

Vi skal nå plotte den faktiske banen til planeten for hver av de tre eksentrisitetene. Vi dekomponerer ligning (5) til x og y komponenter:

$$x(\theta) = r(\theta)\sin(\theta), \qquad y(\theta) = r(\theta)\cos(\theta)$$

Bruk disse ligningene til å plotte $y(\theta)$ mot $x(\theta)$ i det samme tidsintervallet, med de samme eksentrisitetene.

Husk at $r(\theta)$ og θ er arrays, slik at $x(\theta)$ og $y(\theta)$ blir også arrays uten videre, og kan plottes direkte mot hverandre.

Hint: Matplotlib holder ikke aksene proporsjonale i størrelse, slik at en sirkulær bane kan se elliptisk ut. Du kan sette parameteren `matplotlib.pyplot.axis('equal')` for å sørge for at x og y aksene i samme skala. Du kan også inkludere stjernen selv, ved å plotte et gult punkt i (0,0): `matplotlib.pyplot.plot(0, 0, 'yo')`

Filnavn: `planetary_motion.py`

[6] 1 AU er den gjennomsnittlige avstanden mellom jorden og solen.

Oppgave 5.7 - Estimere Plancks konstant

Vi kan ved hjelp av noen få data målinger, estimere Plancks konstant h. For å gjøre dette, tar vi utgangspunkt i Einsteins likning[7].

Einsteins ligning er gitt som

$$E_k = hf - W$$

der h er Placks konstant, f er frekvens til lyset,W er arbeidet som trengs for å løsrive elektronene og E_k er elektronets kinetiske energi.

Anta at vi har fått noen måledata fra ett eksperiment der vi har sendt lys med ulike frekvenser f slik at elektronene fikk en *maksimal* kinetisk energi $E_{k,max}$:

$f/10^{15}$ Hz	1.18	0.96	0.82	0.74
$E_{k,max}/10^{-19}$ J	3.12	1.57	0.8	0.22

Lysfrekvensene f har blitt delt på 10^{15} og verdiene for de maksimale kinetiske energiene $E_{k,max}$ blitt delt på 10^{-19} i tabellen. Når du skal bruke de *egentlige* verdiene for f og $E_{k,max}$, er det viktig å passe på at verdiene skal ganges med henholdsvis 10^{15} og 10^{-19} for at utregningene skal bli riktige!

a)

Lag et program som plotter måledataene som punkter. Verdiene for f og $E_{k,max}$ skal ligge i arrays. Plott verdiene for f langs x-aksen og verdiene for $E_{k,max}$ langs y-aksen.

Hint: For å plotte punkter, er det nok å sende inn et ekstra argument til `plt.plot` om at du ønsker punkter istedenfor linjer. For å få for eksempel røde punkter, kan du sende inn `'ro'` som argument.

b)

Numpy har en funksjon `np.polyfit(x,y,1)` som finner stigningstallet a og konstantleddet b til en rett linje $ax + b = y$ som er *nærmest mulig* de gitte datapunktene for `x` og `y`.

Hvis vi ser på Einsteins likning, har vi nettopp at $a = h$ og $b = -W$ i vårt tilfelle[8]! Vi kan derfor bruke våres målinger til å estimere h og W.

Utvid programmet ditt fra a) slik at den kaller `np.polyfit(`f`,` $E_{k,max}$`,1)` og lagrer verdiene for h og $-W$. La programmet ditt skrive ut verdien av h som den får.

Programmet ditt skal gi en estimert verdi for h som er (i enhet J s)

```
6.50987e-34
```

(som er ganske nærme verdien til $h = 6.626 \times 10^{-34}$ J s!)

c)

Plot datapunktene fra a) sammen med en rett linje $y = ax + b$ der a og b er de estimerte verdiene for henholdsvis h og $-W$.

Filnavn: `estimate_h.py`

[7] Ligningen oppstod da Einstein ville forklare på hvorfor vi har fotoelektrisk effekt, altså hvorfor elektroner kan løsrives fra metaller der lyset har en høy nok frekvens.

[8] hvis du er usikker: sett inn $a = h$,$b = -W$,$x = f$ og $y = E_{k,max}$ i ligningen for en rett linje og se at den er lik med Einsteins likning.

Oppgave 5.8 - Enkel pendel

Bevegelsen til en pendel er avhengig av hvilken vinkel den befinner seg ved en tid t. Her skal vi se på hvordan vi kan programmere en enkel modell for bevegelsen til en pendel. Posisjonene langs x- og y-aksen til pendelen kan uttrykkes ved:

$$x(t) = L\sin(\theta(t))$$
$$y(t) = -L\cos(\theta(t))$$

der $\theta(t)$ er vinkelen til pendelen ved en tid t og L er lengden til pendelen.

Ved å anta at pendelen svinger om små vinkler, kan vi finne at $\theta(t)$ er:

$$\theta(t) = \theta_0\cos(\omega t)$$

der θ_0 er vinkelen pendelen starter ved og $\omega = \sqrt{\frac{g}{L}}$ (der $g = 9.81\,\mathrm{m/s}$) er vinkelhastigheten til pendelen. Vinkelhastigheten er et mål på hvor fort pendelen svinger.

Lag et program som beregner $x(t)$ og $y(t)$ for $N = 1000$ tidspunkter mellom 0 og $T = 1$ sekund. La $\theta_0 = \frac{\pi}{6}$ og $L = 0.75\,\mathrm{m}$. Programmet skal så plotte posisjonene langs x- og y-aksen.

Viktig: Programmet ditt skal ikke bruke noen for eller while loops for å utføre beregningene. Her vil arrays være nødvendige å bruke.

Filnavn: `pendulum.py`

Oppgave 5.9 - Skrått kast

Vi skal nå se tilbake på oppgave 3.4. Der lagde vi et program som simulerer et ballkast avhengig av kastevinkelen θ. I denne oppgaven skal vi se på hvordan ballens bane forandrer seg ettersom hvilken kastevinkel vi har. Ballens posisjon langs x-aksen $x(t)$ og ballens posisjon langs y-aksen $y(t)$ ved en tid t kan modelleres ved

$$x(t) = v_0 t\cos\theta$$
$$y(t) = -\frac{1}{2}gt^2 + v_0 t\sin\theta$$

der $g = 9.81\,\mathrm{m/s^2}$.

Lag et program som genererer 1000 verdier for t mellom 0 og $T = \dfrac{3.5}{v_0\cos\theta}$ s. La $v_0 = 16\,\mathrm{m/s}$ og plott $x(t)$ sammen med $y(t)$ for $\theta = \dfrac{\pi}{6}, \dfrac{\pi}{4}$ og $\dfrac{\pi}{3}$.
Sørg for at programmet ditt bruker arrays og sender demtil funksjoner.

Filnavn: `plot_throw_ball.py`

Oppgave 5.10 - Angulær bølgefunksjon

Kort bakgrunn til oppgaven:I denne oppgaven vil vi se på et utvalg funksjoner. Funksjonene vi vil se på kalles for *Legendre funksjoner* som tar $\cos\theta$ inn som argument. Funksjonene brukes, sammen med andre funksjoner, til å beskrive en bølgefunksjon til en partikkel i tre dimensjoner. Bølgefunksjonen beskriver tilstanden til en partikkel og kan brukes til å se på sannsynligheten for å finne en partikkel i et område.

I denne oppgaven skal vi se på tre funksjoner:

$$P_2^0(\theta) = \frac{1}{2}\left(3\cos^2\theta - 1\right)$$

$$P_2^1(\theta) = 3\sin\theta\cos\theta$$

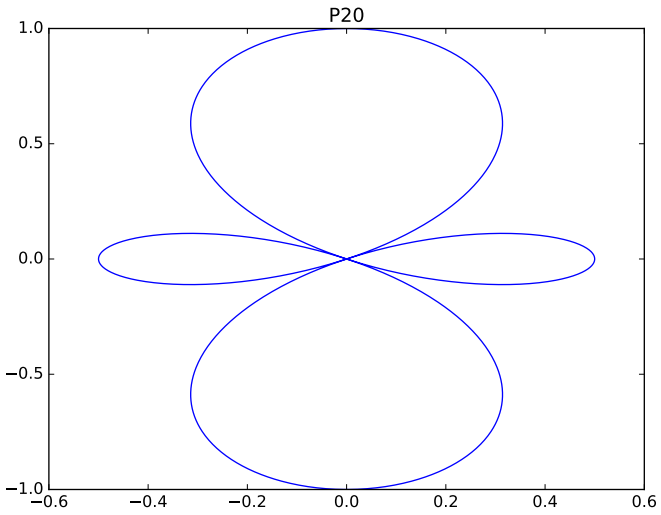
$$P_2^2(\theta) = 3\sin^2\theta$$

Der θ har verdier mellom 0 og π.

Skriv et program der P_2^0, P_2^1 og P_2^2 er funksjoner. La θ være en array som har 1000 jevnt fordelte verdier mellom 0 og π. Programmet skal så gjøre følgende for hvert funksjon P_2^i:

1) Regner ut $R_i = \left|P_2^i(\theta)\right|$
2) Regner ut $a = R_i\sin\theta$ og $b = R_i\cos\theta$ uten å bruke for loop. Her hjelper det å ha definert funksjonene i programmet og la θ være en array.
3) Plotter b langs y-aksen sammen med a langs x-aksen og a speilet om y-aksen. For å få til speilingen, er det nok å plotte $-a$ langs x-aksen sammen med b langs y-aksen. Programmet skal altså lage to grafer i samme plott, og la gjerne grafene ha samme farge.

Hint: Utnytt at funksjoner også kan lagres i lister. Bruk en for loop for å iterere gjennom funksjonene.



Figur 3: Et eksempel på hva du skal få etter å ha plottet $P_0^2(\theta)$.

Filnavn: `angular_wavefunction.py`

Oppgave 6.1 - Solsystem

Følgende fil inneholder informasjon om gjennomsnittlig avstand fra solen, masse, og radius ved ekvator, til en rekke himmellegemer i vårt solsystem.

Object	Distance [km]	Mass [kg]	Radius [km]
Sun	0	1.99e30	695700
Mercury	5.79e7	3.30e23	2440
Venus	1.08e8	4.87e24	6052
Earth	1.50e8	5.97e24	6378
Mars	2.28e8	6.42e23	3396
Jupiter	7.79e8	1.90e27	71492
Saturn	1.43e9	5.68e26	60268
Uranus	2.87e9	8.68e25	25559
Neptune	4.49e9	1.02e26	24764
Pluto	5.90e9	1.46e22	1185

a)

Skriv et program som leser filen `solar_system_data.dat`, og skriver verdiene inn i tre dictionaries `distance`, `mass` og `radius`.

b)

Lag et nytt dictionary, `density`, som inneholder massetettheten til hver av objektene i kg/m^3. Du kan anta at alle planetene er perfekte sfærer. Volumet til en sfære er $V = \frac{4}{3}\pi r^3$.

c) (Valgfri ekstraoppgave)

Kombiner informasjonen i et nøstet dictionary, `solar_system` slik at f.eks. `solar_system[Jupiter][radius]` gir radiusen til Jupiter.

Test at det nye dictionary'et du har laget fungerer som det skal ved å regne ut hvor mange jordkloder det tar å rekke fra solen og til Pluto.

Filnavn: `solar_system_dict.py`

Oppgave 6.2 - Lese og bruke fysiske konstanter

Det er ofte vi har behov for en god del fysiske konstanter for å modellere ulike fysiske systemer.

a)

Lag et program som leser `physics_constant.dat`. Filen har et slikt format slik at navnene (av vilkårlig lengde) står foran et kolon ':'. Deretter, står konstantens symbol, så verdien og til slutt konstantens enhet. Programmet skal lagre konstantene i et dictionary. Konstantenes symbol skal brukes som nøkkel for å hente ut konstantens tilhørende verdi.

b)

Bohr utledet en formel på hvordan en kan modellere energinivåene E_n til et hydrogenatom. Energinivåene er utledet til å være

$$E_n = -\frac{k_e^2 m_e e^4}{2\hbar^2}\frac{1}{n^2}$$

Bruk dictionary-et fra a) for å hente ut de nødvendige konstantene som brukes i modellen, og se om programmet ditt får at

$$\begin{aligned}\frac{k_e^2 m_e e^4}{2\hbar^2} &\simeq 2.18 \times 10^{-18}\,\text{J} \\ &\simeq 13.6\,\text{eV}\end{aligned}$$

der k_e er Coulombs konstant, m_e elektronets masse, e er elementærladningen og $\hbar$ er Plancks reduserte konstant (du finner den i filen som `hbar`).

Filnavn: `constants_hydrogen.py`

Oppgave 6.3 - Dynamisk friksjon

Når et legeme beveger seg på et underlag, virker det en dynamisk friksjonskraft på legemet. Friksjonskraften er dynamisk fordi legemet er i bevegelse.

Den dynamiske friksjonskraften μ_D er

$$\mu_D = N\mu$$

der μ er friksjonstallet mellom materialene legemet og underlaget består av (μ vil variere ettersom hvilken materiale-par vi ser på) og N er normalkraften som virker på legemet[9].

Vi antar legemet beveger seg i horisontal retning. Dette gir at

$$N = mg$$

der m er legemets masse og $g = 9.81\,\text{m}/\text{s}^2$.

Vi skal se på hva den dynamiske friksjonen blir ettersom hvilket materiale legemet og underlaget er laget av.

a)

Lag et program som leser inn filen `friction_coefficients_data.dat` (som du kan finne her: lenke-til-fil) som er en tabell som består av friksjonstall for noen materiale-par.

Den første linjen forteller hvilket material-par vi ser på, og den andre linjen er tilhørende friksjonstall til material-paret.

Programmet skal hente ut og lage separerte lister over material-parene og friksjonstallene. Programmet ditt trenger ikke å ta hensyn til bindestreken mellom materialene (enn så lenge).

b)

Nå skal programmet bruke listene av material-parene og friksjonstallene.

[9] Normalkraften er den kraften som virker på legemet fra underlaget der legemet og underlaget er i kontakt med hverandre.

For hvert materialpar la programmet hente ut materialet til legemet (materialet før bindestreken),materialet til underlaget (etter bindestreken) og tilhørende friksjonstall μ til paret. La programmet beregne den dynamiske friksjonskraften. Kraften skal så skrives til skjerm der det kommer tydelig fram hvilket materiale legemet og overflaten er laget av.

Her lar vi legemet ha masse $m = 2.5\,\text{kg}$.

Filnavn: `dynamic_friction_pair.py`

Oppgave 6.4 - På Jorden

Vi har arbeidet mye med tyngdeakselerasjon $g = 9.81\,\text{m/s}^2$ uten å bekymre oss alt for mye om hvor vi befinner oss på Jorden. Faktisk, så er g avhengig av hvor vi befinner oss (eller; hvor mange meter vi er over havet og ved hvilken breddegrad vi befinner oss)[10]!

I filen `data_different_g.dat` kan du finne verdier for g (avrundet) for et utvalg byer.

a)

Skriv et program som leser filen `data_different_g.dat` og lager et dictionary der verdien av g er lagret for hver by.

For eksempel, er tyngdeakselerasjonen i Stockholm lik $g = 9.818$. Ditt dictionary burde være laget slik at hvis du sender `Stockholm` som nøkkel til det, burde tyngeakselerasjonen $g = 9.818$ bli returnert.

Vi skal anta at en city har enten ett mellomrom i sitt navn (for eksempel San Francisco), eller ingen (for eksempel Paris).

Den siste linjen som inneholder kun '`-`'s markerer slutten på listen over byene.

Hint: Bruk `split()` for hver linje for å få en liste av strings som opprinnelig var separert av mellomrom. Dersom et navn inneholder ett mellomrom, så vil listen bli 1 enhet lengre enn listen med navn uten mellomrom. En kan derfor sjekke lengden til listen for å se om bynavnet har et mellomrom eller ei.

Slicing vil også være veldig nyttig for å hente ut verdier for g.

b)

Utvid programmet ditt fra a) slik at den leser filen `on_Earth.dat`. Filen inneholder flere linjer, hvorav hver av linjene består av to byer seperert med `to`.

For hver by i `on_Earth.dat`, må programmet ditt hente ut navnet av hver by og bruke dictionary-et fra a) for å skrive ut tyngdeakselerasjonen til byene sepearert med `to` (i samme format som i `on_Earth.dat`, bare med tilhørende verdier av g for hver by).

Dette viser hvordan utskriften skal se ut:

```
9.813 to 9.8
9.805 to 9.816
9.82 to 9.78
```

[10] Dette er på grunn av sentrifugalkraften - en kraft som vi oppfatter som en tilstedeværende kraft på grunn av Jordens rotasjon

Den siste linjen i filen markerer slutten. Du kan anta at ingen bynavn begynner på '`On`'.

Hint: Igjen er `split()` din venn. Utnytt antakelsen at bynavnene består enten av ett mellomrom eller ingen, ingen by har bynavn som begynner på '`On`' og det er alltid en `to` som separerer byene.

For den andre byen (etter '`to`'), kan en bruke `' '.join(line[start:])` for å hente ut dets hele navn. Indeksen `start` er avhengig om hvorvidt den første byen har et navn med eller uten ett mellomrom.

Filename: `different_g.py`

Oppgave 7.1 - Planet-klasse

a)

Du skal skrive en klasse `Planet`, som tar inn informasjon om en planets *navn*, *radius* og *masse*, og lagrer verdiene. Inkluder en metode `density`, som returnerer massetettheten til planeten, i $\mathrm{kg/m^3}$, og en metode `print_info`, som printer all kjent informasjon om planeten, inkludert massetettheten. (Du kan kalle `density` metoden fra `print_info`).

b)

Lag en instans av klassen kalt `planet1`, som representerer Jorden (sett gjerne navnet til Earth"). Legg til en ny "attributetil instansen, kalt `population`, med en verdi `7497486172`.

Legg ved følgende linje til slutten av programmet ditt. Hvis du har implementert alt korrekt, skal du få printen vist under:

```
print planet1.name, "has a population of ", planet1.population
```

```
>>> Earth has a population of 7497486172
```

Filnavn: `Planet.py`

Oppgave 7.2 - Coulombs lov

Coulombs lov forteller oss om hvilken kraft som virker mellom to ladede punktlegemer [11].

Vi skal nå se på hvordan vi kan lage en enkel modell til å finne kraften som virker mellom to ladede partikler.

Coulombs lov er definert på følgende måte:

$$F = k_e \frac{q_1 q_2}{r^2}$$

der $k_e = 8.988 \times 10^9\,\mathrm{Nm^2\,/C^2}$, q_1 er ladningen til et punktlegeme, q_2 er ladningen til det andre punktlegemet og r er avstanden mellom partiklene.

a)

Definér en klasse som tar inn partikkelens posisjon (som en array bestående av punktlegemets x- og y-koordinat) og ladning q i konstruktøren.

Klassen skal også inneholde en funksjon som tar inn en annen partikkel som parameter og beregner kraften fra Coulombs lov som virker mellom dem. Kraften i absoluttverdi skal så returneres.

Hint: For å beregne avstanden r mellom partiklene, kan `np.linalg.norm` brukes.

[11] Et punktlegeme er et legeme uten utstrekning. En kan tenke på punktlegeme som noe som finnes, men er såpass liten at det tar ingen plass. Små legemer, som f.eks elementærpartikler pleier vi vanligvis å kalle for punktlegemer. For legemer som har utstrekning og er kuleformede kan vi også anvende denne loven.

b)

Lag så en testfunksjon *utenfor* klassen for å teste om din implementasjon av Coulombs lov gir ønsket resultat. La så programmet kalle på testfunksjonen.

En mulig test som programmet ditt kan gjøre, er å lage to partikler som har 30 mm avstand mellom seg, hvorav den ene partikkelen har ladning -1.602×10^{-19} C og den andre 1.602×10^{-19} C.

Da skal funksjonen som beregner kraften som virker mellom partiklene gi at

$$F = 2.565\,833\,688 \times 10^{-15}\,\mathrm{N}$$

Filnavn: `Particle_Coulomb.py`

Oppgave 7.3 - Uidentifisert flyvende objekt

a)

Du skal skrive en klasse, `ObjectMovement`, som skal regne på bevegelsen til et objekt som flyr fritt gjennom luften ved jordoverflaten. Vi ser bort i fra luftmotstand, og regner i to dimensjoner - en horisontal akse x, og en vertikal akse y.

Start klassen med en konstruktør som lagrer objektets initialposisjon `(x0,y0)`, og initialhastighet `(vx0,vy0)`. Inkluder også akselerasjonen ($a = -9.81\,\mathrm{m/s^2}$) her, gjerne som en keyword-variabel.

Gi klassen to metoder `position` og `velocity` som tar inn et tidspunkt t, og returnerer objektets posisjon eller hastighet på dette tidspunktet.

$$p = p_0 + v_0 t + \frac{1}{2}at^2, \qquad v = v_0 + at$$

Husk at du kan dekomponere bevegelsen i horisontal og vertikal retning, og bruke bevegelseslikningene langs hver akse. Her er det greit å huske at akselerasjonen bare virker i y-aksen.

Skriv også en test-funksjon `test_pos_vel` som sjekker om posisjon og hastighet fra metodene `position` og `velocity` ved et gitt tidspunkt stemmer med eksakte verdier du har regnet ut med kalkulator (innenfor en toleranse).

b)

Fordi objektet bare er under påvirkning av tyngdekraften, som er en konservativ kraft, skal summen av *kinetisk* og *potensiell* energi være bevart. Kinetisk energi er gitt som $E_k = \frac{1}{2}mv^2$, og potensiell energi er (på jordens overflate) gitt som $E_p = mgy, \quad g = 9.81\,\mathrm{m/s^2}$.

Husk at $v = \sqrt{v_x^2 + v_y^2}$.

Skriv en funksjon `test_energy_conservation` som regner ut E_k og E_p på to ulike tidspunkt, og sjekker om summen av energiene er identisk for de to tidspunktene. Sett $m = 1$.

Filnavn: `UFO.py`

Oppgave 7.4 - Løpere på ulike helninger

Vi skal nå lage en klasse som representerer løpere som sprinter ned ulike bakker med ulike vinkler.

a)

Løperne har masse m kg og en startfart på v_0 m/s. Massen m, startfarten v_0 og helningsvinkelen θ skal tas inn som parametre til konstruktøren. Lag klassen som representerer en sprinter og lag en sprinter-instans med masse $m = 80$ kg, startfart $v_0 = 5$ m/s og $\theta = 30°$.

b)

Utvid klassen fra a) slik at den inneholder en funksjon `__str__`(denne funksjonen kalles automatisk når vi prøver å printe instansen).
Funksjonen skal returnere en string som inneholder løperes masse, startfart og helningsvinkel til bakken løperen sprinter på. Pass på at det kommer tydelig fram i string-en hva hver enkelt verdi representerer.

Dersom vi prøver å printe løper-instansen fra a), kan vi f.eks få:

```
Sprinter with
 mass: 80 kg
 initial velocity: 5 m/s
 angle: 30 degrees
```

c)

Nå skal også klassen ha en funksjon som regner ut hvor lang tid løperen vil bruke for å løpe ned hellingen d meter.
Vi forenkler kraften F_d som driver løperen fram ved å sette $F_\mathrm{d} = 400\,\mathrm{N}$.
Utvid klassen med en funksjon som tar inn avstanden d som parameter. Tiden T som løperen vil bruke for å løpe d meter kan en vise er

$$T = -\frac{v_0}{g\sin\theta + \frac{1}{m}400} + \frac{\sqrt{v_0^2 + 2d\left(g\sin\theta + \frac{1}{m}400\right)}}{g\sin\theta + \frac{1}{m}400}$$

Bruk denne funksjonen og skriv ut hvor lang tid løperen fra deloppgave a) vil bruke.

Filnavn: `Runner.py`

Oppgave 7.5 - Massesenter

Når vi jobber med med flere legemer, er det ofte nyttig å jobbe med *massesenteret* mellom legemene. Massesenteret er en posisjon som ofte brukes som et referansepunkt.

Hvis vi har N legemer der j-te legeme har masse m_j og posisjon $\vec{r}_j$, er massesenteret definert som:

$$\vec{R} = \frac{m_1\vec{r}_1 + m_2\vec{r}_2 + \cdots + m_N\vec{r}_N}{m_1 + m_2 + \cdots + m_N}$$

I denne oppgaven kommer vi til å jobbe med arrays med lengde 2 som representerer et legemets posisjon. Første element i arrayet vil representere legemets x-koordinat, mens andre element i arrayet vil representere legemets y-koordinat.

Vi skal nå se på massesenteret blant et utvalg partikler.

a)

Definér en partikkel klasse som tar inn en partikkels posisjon og masse i konstruktøren.

b)

Opprett fem partikkel-instanser der j-te partikkel har posisjon $r_j = (j, 2 \cdot j)$ og masse $m_j = j \cdot 10^{-30}$ kg.

Hint: En måte en kan opprette instansene på, er å opprette dem gjennom en for loop og lagre dem i en liste

c)

La programmet ditt finne massesenteret mellom de fem partiklene du opprettet i b).

Filnavn: `center_of_mass.py`

Oppgave 8.1 - Energikonservering

Nå skal vi se på hvordan vi kan bruke generering av tilfeldige tall for å bekrefte et fysisk fenomen. Dette fenomenet som vi skal se på, er *energikonservering*. Energikonservering brukes ofte når vi løser problemer i fysikk. Et legemes totale energi E kan deles inn i to hovedkategorier: kinetisk energi E_k og potensiell energi E_p. Det finnes også flere former for energi, men vi ser bort i fra disse i denne oppgaven.

Vi har da at legemets totale energi E kan skrives som $E = E_k + E_p$. Dersom vi har konservering av energi, så betyr det at for *uansett* tid t vi måler legemets energi ved vil energien E være den samme. Vi kan skrive dette som

$$E_0 = E_1$$
$$E_{k,0} + E_{p,0} = E_{k,1} + E_{p,1}$$

der E_0 og E_1 er de totale energiene målt ved de ulike tidene t_0 og t_1.

Kinetisk energi og potensiell energi for et legeme som blir kun påvirket av gravitasjonskraften er

$$E_k = \frac{1}{2}mv(t)^2$$
$$E_p = mgy(t)$$

Vi ser på en ball som blir kastet rett opp. Høyden $y(t)$ og farten $v(t)$ til ballen, $y(t)$, ved en tid t kan vi finne ved

$$y(t) = -gt^2 + v_0 t$$
$$v(t) = -gt + v_0$$

der $g = 9.81\,\mathrm{m/s^2}$ og $v_0\,\mathrm{m/s}$ er kastefarten.

Lag en testfunksjon som tar inn N, m og v_0 som parametre og sjekker om vi har energikonservering. La funksjonen generere N tilfeldige verdier for tiden t_0 og N tilfeldige verdier for tiden t_1. Verdiene skal ligge mellom 0 og $\frac{v_0}{g}$.
For alle verdier av t_0 skal programmet regne ut E_0, og tilsvarende skal programmet regne ut E_1 for alle verdier av t_1.
Sjekk så om verdiene er omtrent like, og bruk `assert` for å avbryte programmet og skrive ut passende feilmelding dersom vi ikke har energikonservering.

Sett $N = 100$, $m = 0.057\,\mathrm{kg}$ og $v_0 = 17\,\mathrm{m/s}$ og kall på testfunksjonen for disse parametrene.

Hint: Mye av koden kan vektoriseres - se det nederste eksempelet på s.510 i boken for hvordan testing kan vektoriseres.

Filnavn: `check_energy_conservation.py`

Oppgave 8.2 - Tilfeldig nedbrytning

Vi har tidligere studert hvordan en mengde radioaktivt materiale nedbrytes over tid, fra en rent analytisk formel, gitt som

$$N(t) = N_0 \mathrm{e}^{-t/\tau} \tag{6}$$

Med vår nye kjennskap til tilfeldige tall kan vi løse problemet på en mer numerisk måte, ved å modellere hvert atom individuelt.

Hvert atom i et radioaktivt materiale har en sjanse p for å nedbrytes hvert sekund. Vi kan modellere materialet som en array, hvor hvert element representerer et atom. Vi gir hver atom en verdi 1, som representerer at atomet ikke er nedbrutt, eller en verdi 0.

Vi skal se på et isotop av nitrogen, *nitrogen-16*, hvor hvert atom har en sjanse $p = 0.0926$ (eller 9.26%) for å nedbrytes hvert sekund.

a)

Vi skal først studere det første sekundet av nedbrytningen. Lag et array av lengde 40 med verdier 1 (som representerer 40 atomer som ikke er nedbrutt), og loop over hvert element i arrayet for å bestemme om hvert atom nedbrytes til 0 eller ikke. Du kan gjøre dette ved å sammenligne tallet p med et tilfeldig generert tall i intervallet $[0,\ 1]$.

Print det nye arrayet, og bekreft at noen av atomene faktisk ble nedbrutt til 0.

b)

Repeter prosessen over 20 sekunder ved å legge en til for loop rundt hele koden.

Implementer også et nytt array som lagrer hvor mange atomer som gjenstår etter hver iterasjon. Du kan gjøre dette ved med å bruke funksjonen `numpy.sum()` på arrayene[12]. Få også med tidspunkt $t = 0$ i arrayet, som er før første iterasjon i loopen.

c)

Plot mengden gjenværende atomer over tid. Inkluder også den analytiske løsningen for nedbrytingen av nitrogen-16, gitt av ligning (6). Sett N_0 lik antallet atomer, og $\tau = 10.3\,\text{s}$ (som er et resultat av valget av p).

Prøv å øke antallet atomer i simuleringen, og (forhåpentligvis) observer at den numeriske løsningen blir mer og mer lik den analytiske.

Filnavn: `random_decay.py`

Oppgave 8.3 - Optimale skytevinkler

Du har laget en liten ballkanon som skal treffe et målt område på en vegg (har du løst 3.3 i dette heftet tidligere, kan du godt gjenbruke og modifisere koden i denne oppgaven). Området begynner ved en høyde h_0 og slutter ved en høyde h_1 på veggen. Høyden til ballen kan modelleres ved

$$y(t) = -\frac{1}{2}gt^2 + v_0 t \sin\theta$$

der $g = 9.81\ \text{m/s}^2$, v_0 m/s skytefarten og θ skytevinkelen. Ballen treffer veggen ved en tid $T = \dfrac{b}{v_0 \cos\theta}$, der b er avstanden (i meter) mellom kanonen og veggen.

a)

Lag et program som genererer N verdier av θ ved bruk av Pythons `uniform(0,π)` eller Numpys `uniform(0,π,N)`.
Verdiene for θ skal sendes inn som parameter til en funksjon som sjekker om kanonen treffer innenfor området ved de gitte θ-ene. De θ-ene som gir at ballen treffer treffer det malte området skal bli lagret i en liste eller array. Listen eller array-et skal så returneres av funksjonen.

[12] Denne funksjonen summerer opp alle verdiene i arrayet. Dette betyr at nedbrutte atomer teller som 0, og gjenværende som 1.

Sett $N = 1000$, $h_0 = 3\,\text{m}$, $h_1 = 3.25\,\text{m}$, $b = 3.5\,\text{m}$ og $v_0 = 25\,\text{m/s}$ og kjør programmet for disse parametrene.

b)

Bruk listen eller array-et som ble returnert fra funksjonen i a) for å finne gjennomsnittet θ' av de vinklene som fikk ballen til å treffe det avgrensede området. Hvis listen er tom, skal programmet generere nye N verdier for θ helt til den returnerte listen eller array-et ikke er tom. Dette kan for eksempel gjøres ved å ha en while loop som vil kjøre så lenge listen er tom.

La programmet gjøre det samme som i a) med den forskjell i at verdiene for θ blir generert ved `random.gauss(`θ'`,0.05)` eller `np.random.uniform(`θ'`,0.05,`N`)`.

La programmet ditt skrive ut hvor mange vinkler fra a) og fra denne deloppgaven som traff området. Hvordan er antallet vinkler som treffer området sammenlignet med antallet som ble generert i a)?

Kommentarer til resultatet i b): Verdiene av θ i a) ble generert slik at verdiene mellom 0 og π var like sannsynlige. I b) derimot, ble verdiene generert ved å bruke en *normal fordeling* sentrert om θ'.

Normalfordelingen har en bjelleform og forteller oss hvor sannsynlig det er for at vi trekker verdier i et område om θ'. Verdien til θ' vil være mest sannsynlig å få, mens sannsynligheten til å få andre verdier vil minke jo større differansen er mellom verdiene og θ. Hvor brått sannsynligheten vil synke, er også avhengig av en ekstra verdi, kalt standardavviket, som i dette tilfellet er 0.05 (som er funnet ved litt eksperimentering).

Jo større standardavvik en bruker, jo mest sannsynlig blir de andre verdiene, og mindre standardavvik vil gi at de andre verdiene blir mindre sannsynlige å få.

Filnavn:`optimal_angles_shoot.py`

Oppgave 8.4 - Varm gass

I denne oppgaven skal vi studere hastigheten til partikler i en varm gass. Temperatur er egentlig bare tilfeldig uorganisert bevegelse. Hvis gassen står stille, vil selvsagt den totale bevegelsen til alle partiklene i gassen være 0, men det forekommer fortsatt mye intern bevegelse.

Partikler i en gass med absolutt temperatur T har hastigheter som er normalfordelt (Gaussisk fordelt) rundt et gjennomsnitt på 0, med et standardavvik [13] $s = \sqrt{\frac{kT}{m}}$, hvor $k = 1.38 \times 10^{-23}\,\text{m}^2\,\text{kg}\,\text{s}^{-2}\,\text{K}^{-1}$ er Boltzmann's konstant. Denne fordelingen gjelder individuelt for hastigheten langs hver akse, (v_x, v_y, v_z).

Vi kan generere tilfeldig normalfordelte tall med numpyfunksjonen `numpy.normal(m,s,shape)`, hvor `shape` er dimensjonene til arrayet at tilfeldige tall. For å representere `N` partikler med normalfordelt hastighet langs hver akse, trenger vi et array på formen `shape = (N,3)`. Dette betyr at hvert element i arrayet representerer en enkelt partikkel, som i seg selv er en array av 3 elementer, som igjen representerer hastigheten langs hver akse.

a)

Set $N = 20$, $m = 10^{-22}\,\text{kg}$, $T = 300\,\text{K}$, bruk numpyfunksjonen til å lagre et array av normalfordelte hastigheter.
Bruk `random.choice()` funksjonen til å velge ut en tilfeldig av de 20 partiklene, og print hastighetene til partikkelen til terminalen.

[13] Standardavviket representerer bredden på fordelingskurven, og et høyere standardavvik bestyr større hastighet, og høyere temperatur.

b)

Den totale kinetiske energien til N partikler i en gass av temperatur T er gitt som

$$E_k = \frac{3}{2}kTN$$

En alternativ metode for å regne ut den kinetiske energien til gassen er å summere opp den kinetiske energien til hver enkelt partikkel, som er gitt som

$$E_k = \frac{1}{2}mv^2$$

Loop over hver partikkel, og beregn deres kinetiske energi ved å sette absoluttverdien av hastigheten, $|v| = \sqrt{v_x^2 + v_y^2 + v_z^2}$, inn i formelen over. Sammenlign resultatet med det du fikk fra den øverste formelen.

Filnavn: `gaussian_velocities.py`

Oppgave 8.5 - Regndråper

Vi skal se på *terminalfarten* til mange regndråper. Terminalfarten er farten en regndråpe har når tyngdekraften trekker dråpen ned like mye som den møter luftmotstand.

For en regndråpe med radius r meter der vi antar dråpen har en perfekt kuleform, kan vi regne ut at terminalfarten v_T (i $\mathrm{m/s}$) vil være

$$v_T = \sqrt{\frac{8r\rho_v g}{3\rho C}}$$

der $r\,\mathrm{m}$ er radiusen til dråpen, $\rho_v = 1\,\mathrm{g/c^3m} = 1000\,\mathrm{kg/m^3}$ er den omtrentlige tettheten til vann, $g = 9.81\,\mathrm{m/s^2}$, $\rho = 1.293\,\mathrm{kg/m^3}$ er omtrentlige tettheten til luft og $C = 0.47$ betegner hvor mye motstand regndråpen utgjør på sin omgivelse (i dette tilfellet: luften).

a)

I denne deloppgaven skal vi *ikke* bruke Numpy-modulen.

Lag et program som generer $N = 100000$ ulike regndråper representert ved tilfeldige verdier for radius i intervallet $[1\,\mathrm{mm}, 6\,\mathrm{mm}) = [10^{-3}\,\mathrm{m}, 6 \cdot 10^{-3}\,\mathrm{m})$.

Programmet skal finne ut hvor lang tid det tar for å generere de N regndråpene og beregne gjennomsnittet av terminalfartene til dem.

Programmet skal tilslutt skrive ut den gjennomsnittlige terminalfarten og hvor lang tid programmet brukte.

b)

Utvid programmet ditt slik at samme beregninger som i a) utføres, men nå med bruk av Numpy-modulen og vektorisering. Skriv ut den gjennomsnittlige terminalfarten sammen med tidsbruket programmet brukte ved vektorisering og bruk av Numpy-modulen.

I denne deloppgaven vil du også kunne se en betydelig forskjell i tidsbruk sammenlignet med tidsbruket fra a).

Hint: For å ta tiden i sekunder til et program, kan du bruke `time`-modulen. Et eksempel på en kode som finner ut hvor lang tid programmet bruker for å skrive 'Hello, world!' til skjerm, er:

```
import time

time_start = time.time()             # To start taking time
print "Hello, world!"
time_used = time.time() - time_start # Find how how time we have used

print "Time used: %g seconds"%time_end
```

Filnavn: `raindrops.py`

Oppgave 9.1 - Lineær akselerasjon

Filen ...INSERT FILE REFRENCE...`constant_acceleration_class.py` inneholder en klasse `ConstantAcceleration` som regner på posisjon og hastighet til et objekt i endimensjonal bevegelse med konstant akselerasjon. Konstruktøren lagrer initialposisjonen, og initialhastigheten, samt akselerasjonen. Klasse-kallet returnerer posisjonen på et tidspunkt `t`, og det finnes en metode `velocity` som returnerer hastigheten på et tidspunkts `t`.

a)

Hensikten med denne oppgaven er lage en ny klasse `LinearAcceleration` som arver funksjonaliteten til `ConstantAcceleration`. Klassen skal også kunne håndtere tilfeller med lineær akselerasjon på formen $a = a_0 + jt$. Konstanten j kalles "jerk", og er endring i akselerasjon over tid. De nye bevegelsesligningen ser nå ut som

$$x(t) = x_0 + v_0 t + \frac{1}{2} a_0 t^2 + \frac{1}{6} j t^3$$

$$v(t) = v_0 + a_0 t + \frac{1}{2} j t^2$$

Lag klassen `LinearAcceleration`, som arver funksjonaliteten til `ConstantAcceleration`, men også tar inn variabelen j. Bruk kode fra den opprinnelige klassen så ofte som mulig.

Filename: `Jerk.py`

Oppgave 9.2 - Faststoff

I denne oppgaven skal vi se på hvordan vi kan lage en enkel modell for et vilkårlig fast stoff, for så et spesifikt fast stoff ved hjelp av arv.

a)

Lag en klasse `Solid` som tar inn legemets volum i konstruktøren, og lagrer verdien for legemets volum. Definer så en funksjon som regner ut legemets massetetthet der legemets masse gis som parameter til funksjonen. Massetettheten er definert som legemets masse delt på dets volum.

b)

Lag en klasse `Iron` som er en underklasse av `Solid`. Konstruktøren skal ta inn legemets (som er laget av jern) volum og masse.

Definer en funksjon i `Iron` som kaller på funksjonen i `Solid` for å regne ut og returnere legemets massetetthet.

c)

Definer en testfunksjon som oppretter en instans av klassen fra b) der volumet er 0.1 m^3 og massen er 787 kg. Funksjonen skal teste for om tettheten blir $7870\,\mathrm{kg}\,/\mathrm{m}^3$. Husk å kalle på testfunksjonen!

Filnavn: `Solid.py`

Oppgave 9.3 - Treghetsmoment om massesentre

Massen til et legeme kan sees på som et mål på hvor vanskelig det er å få legemet til å forandre dets hastighet. Vi har et tilsvarende mål når det gjelder å rotere et legeme. Der ser vi på legemets *treghetsmoment* om sitt eget massesenter [14].

a)

Definer en klasse som representerer et geometrisk objekt. Konstruktøren skal kun ta inn legemets masse M.

Lag en instans av denne klassen med $M = 5$ kg.

b)

Lag en klasse som arver fra klassen du definerte i a). Denne klassen skal representere et legeme som er en sylinder. I tillegg til legemets masse M, skal også konstruktøren ta inn radiusen R som parameter.

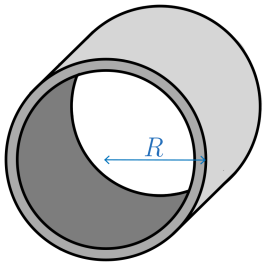
Definér en funksjon i denne klassen som regner og returnerer sylinderens treghetsmoment. Treghetsmomentet I til en sylinder med masse M og radius R om sitt massesenter er funnet til å være

$$I = \frac{1}{2}MR^2$$

Lag en instans av denne klassen med $M = 5$ kg og $R = 0.75$ m og skriv ut dets treghetsmoment.

c)

Lag en klasse som arver fra klassen i b). Denne klassen skal representere et sylinderskall med masse M og radius R.



Figur 4: Illustrasjon av et sylinderskall med radius R

Denne klassen skal også beregne og returnere syldinerskallets treghetsmoment. Men i dette tilfellet kan programmet ditt bruke det som allerede har blitt regnet fra klassen i b). Det er nemlig slik at treghetsmomentet til et sylinderskall om sitt massesenter er funnet til å være

$$I = MR^2$$

Derfor kan denne klassen regne ut sitt treghetsmomentet ved å først regne ut treghetsmomentet til en sylinder, for så gange resultatet med 2.

Lag en instans av denne klassen med $M = 5$ kg og $R = 0.75$ m og skriv ut dets treghetsmoment.

[14] Massesenteret til et legeme er en posisjon som er avhengig av legemets fordeling av masse og form

Bemerkning: Her er det meningen at du skal skrive ganske så lite kode i denne deloppgaven. Det blir veldig lite kode dersom du utnytter arv til det fulle!

Filnavn: `Moment_of_inertia.py`

Oppgave E.1 - Koke opp vann

Newtons avkjølingslov beskriver hvordan temperaturen $T(t)$ til et objekt forandrer seg over t minutter. Den er formulert slik:

$$\frac{\mathrm{d}T(t)}{\mathrm{d}t} = -k(T(t) - T_o)$$

der T_o er temperaturen til omgivelsen og k er en konstant som forteller hvor mye temperaturen endrer seg per minutt.

Vi skal se nærmere på hvordan temperaturen til vann ved 15 °C forandrer seg når det kokes opp i en ganske så gammel og slitt vannkoker. Her skal vi bruke Newtons avkjølingslov som en veldig forenkling av hvordan temperaturen vil forandringe seg. Da er $T(0) = 15\,°\mathrm{C}$. Vi setter $k = 0.2$.

a)

Vannkokeren får en temperatur lik 100°C med en gang den startes opp. Vi har altså at $T_o = 100\,°\mathrm{C}$ for alle t minutter.

Lag et program som bruker `ODESolver` for å regne ut temperaturen til vannet $T(t)$. Bruk $N = 1000$ likt fordelte verdier for t som er mellom 0 og 15.

b)

Som du kanskje så i a), så tar det ganske lang tid før vannet når 100°C. En venn av deg prøver å ordne vannkokeren, men uheldigvis greier å gjøre den dårligere. Du gjør noen målinger for å se hvor ille vannkokeren din har blitt, og merker at temperaturen på vannkokeren endrer seg nå slik:

$$T_o(t) = 20 \sin\left(\frac{\pi}{3}t\right) + 80$$

Utvid programmet ditt fra a) slik at den løser med den nye modellen for hvordan temperaturen i vannkokeren er, .

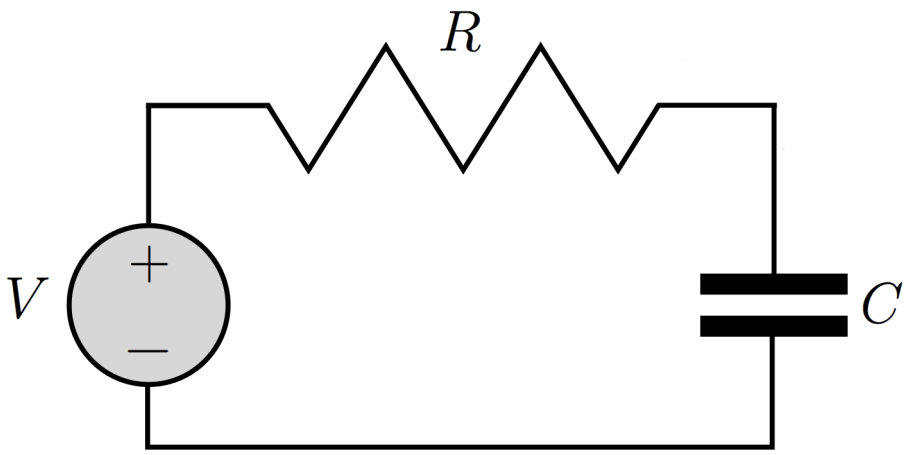
La programmet ditt plotte temperaturen til vannet $T(t)$ fra a) sammen temperaturen $T(t)$ programmet ditt får i denne deloppgaven.

Filnavn: `boiling_water.py`

Oppgave E.2 - RC krets

Hensikten med denne oppgaven er å studere endringen i elektrisk ladning Q holdt av en kondensator over tid.

Figur 5 viser en RC-krets, med en kondensator, en motstand og et batteri. Vi skal konsentrere oss om kondensatoren, som består av to parallelle metalplater. Når vi kobler opp batteriet, vil disse platene lades opp. Når vi kobler fra batteriet, vil kondensatoren utlades ved at strøm går tilbake gjennom kretsen, helt til $Q = 0\,\mathrm{C}$.



Figur 5: Illustrasjon av en RC-krets.

Batteriet har en konstant spenning V (målt i Volt) mens det er på. Motstanden har en resistans R (målt i Ohm) som sier oss hvor godt den hindrer strøm. Kondensatoren har en kapasitans C (målt i Farad).

Ladningen Q holdt av kondensatoren på et tidspunkt t er gitt av differensialligningen

$$Q'(t) = \frac{V}{R} - Q(t)\frac{1}{RC} \tag{7}$$

a)

Bruk `ODESolver` til å løse for $Q(t)$ over de første 10 sekundende, med 101 tidssteg. Sett batteriets spenning til $V = 8\,\text{Volt}$, kondensatorens kapasitans til $C = 2 \times 10^{-8}\,\text{Farad}$, og motstandens resistans til $R = 10^8\,\text{Ohm}$. Kapastitatoren har ingen ladning, $Q_0 = 0$, ved $t = 0$. Bruk både Forward Euler og Runge Kutta 4, og plott resultatene fra begge løsningene sammen med den analytiske løsningen

$$Q(t) = (Q_0 - CV)\mathrm{e}^{-t/RC} + CV \tag{8}$$

Studer hvor mye du må redusere antallet tidssteg for å se forskjellen mellom de numeriske løsningene og den analytiske.

b)

Batteriet blir ustabilt på tidspunktet $t = 10\,\text{s}$, og spenningen begynner å oscillere som en sinuskurve. Ved $t = 20\,\text{s}$ slutter batteriet å virke. Vi kan implementere dette i løsningen vår ved å la spenningen fra batteriet være en funksjon av tid, $V(t)$. Vi ser forsatt på differensialligningen *ligning* (7), men nå er spenningen gitt som

$$V(t) = \begin{cases} 8, & t < 10 \\ 4\sin(t) + 4, & 10 < t < 20 \\ 0, & t > 20 \end{cases} \tag{9}$$

Plott den nye løsningen $Q(t)$ over 30 sekunder med Runge Kutta 4 og 101 tidssteg.

Filnavn: `RC.py`

Oppgave E.3 - Ballkast med luftmotstand

I denne oppgaven skal vi simulere et ballkast der vi inkluderer luftmostand. For å gjøre det, skal vi bruke `ODESolver.py`.

Vinden beveger seg i samme retning som ballen blir kastet.

Ligningene vi skal løse for å finne ballens posisjon er:

$$\begin{aligned}\frac{\mathrm{d}x}{\mathrm{d}t} &= v_x \\ \frac{\mathrm{d}v_x}{\mathrm{d}t} &= -\frac{1}{2m}\rho C\pi R^2(v_x - w)^2\end{aligned}$$

$$\begin{aligned}\frac{\mathrm{d}y}{\mathrm{d}t} &= v_y \\ \frac{\mathrm{d}v_y}{\mathrm{d}t} &= -g\end{aligned}$$

der $C = 0.47$ er et mål på hvor mye motstand ballen gjør på luften, R er radiusen til ballen, m er ballens masse, v_x er ballens fart langs x-aksen, w er farten til vinden og $g = 9.81\,\mathrm{m/s^2}$.

Initialbetingelsene for denne modellen er

$$\begin{aligned}x(0) &= 0 \\ v_x(0) &= v_0\cos\theta \\ y(0) &= 0 \\ v_y(0) &= v_0\sin\theta\end{aligned}$$

der θ er kastevinkelen og v_0 kastefarten (i m/s).

a)

Definér en klasse som holder på informasjon om dette problemet. Klassen skal ha samme struktur som klassen `Problem` på side 790 i boken (eller side 745 hvis du bruker 4. utgave).
Klassen i denne oppgaven skal ha en konstruktør som tar inn initialbetingelsene i form av en liste, radiusen R til ballen, ballens masse m og vindfarten w.

Klassen skal også ha en `__call__`-funksjon som returnerer listen

$$\texttt{[vx,}-\frac{1}{2m}\rho C\pi R^2(v_x - w)^2\texttt{,vy,}-g\texttt{]}$$

der `vx` og `vy` er funnet på samme måte som i eksempelet på s. 790 (eller 745) i boken.

b)

Vi skal nå se på hvordan vindfarten w påvirker ballens bane over $T = 0.5$ sekunder.

Anta vi kaster en ball med radius $R = 0.03275\,\mathrm{m}$ og masse $m = 0.057\,\mathrm{kg}$ med en vinkel $\theta = \frac{\pi}{4}$ og kastefart $v_0 = 10\,\mathrm{m/s}$. Vi lar også $x(0) = 0$ og $y(0) = 0$.

Vi skal se på vindfartene $w = -10, -5, 0, 5,$ og 10. For hver vindfart w skal programmet:

1) Lage en instanse av klassen definert i a) der verdiene for R,m, w og listen av initialbetingelsene sendes inn. Listen av initialbetingelsene `U0` kan ha følgende form:

   `U0` = `[x0,vx0,y0,vy0]` der `x0` $= x(0)$, `vx0` $= v_0\cos(\theta)$, `y0` $= y(0)$ og `vy0` $= v_0\sin(\theta)$.

2) La `ODESolver` løse for posisjonene. Her kan du velge hvilken metode du vil bruke fra `ODESolver` for å løse ligningssettet. Pass også på å sende inn mange nok verdier for tiden til `ODESolver` sin funksjon `solve` for å presise nok resultater.

3) Send de funnede fra posisjonene fra punkt 2) til `plt.plot(x,y)`. Pass på å hente ut de riktige verdiene fra matrisen som `ODESolver` sin `solve` returnerer!

Når programmet er ferdig å ha gått gjennom alle verdiene for w, skal programmet kalle på `plt.show()` for å vise fram ballens bane for ulike w.

Viktig: Når du skal plotte flere grafer i ett plott, er det viktig å kunne skille på dem. For å gjøre det, kan vi bruke en liste av strenger som forteller hver som er unikt ved hver graf. Listen kan så sendes som parameter til `plt.legend` *før* `plt.show()` kalles. I vårt tilfelle kan vi gjøre følgende:

```
# Other necessary calls
labels = []
for w in [-10,-5,0,5,10]:
    # Here we will solve for x and y using ODESolver
    labels.append("w = %g"%w)
    plt.plot(x,y)

plt.legend(labels)
plt.show()
```

for å kunne vite hva vindfarten var ved hver bane.

Filnavn: `throw_air_resistance.py`

Oppgave E.4 - Planetbane

I denne oppgaven skal vi studere banen til jorden rundt solen ved hjelp av andre ordens ODEs. Fordi jordens bane rundt solen er flat, kan vi modellere dette i to dimensjoner, x og y.

Newtons gravitasjonslov forteller oss styrken på gravitasjonskreftene mellom to legemer. Hvis vi dekomponerer dette i x og y retninger, og setter det inn i Newtons andre lov, kan vi løse for akselerasjonen:

$$\frac{d^2x}{dt^2} = -G\frac{M_{sol}x}{x^2+y^2} \qquad \frac{d^2y}{dt^2} = -G\frac{M_{sol}x}{x^2+y^2} \tag{10}$$

Her er M_{sol} massen til solen, G er newtons gravitasjonskonstant, og x og y er avstanden fra jorden til solen i x og y retning.

Vi skal bruke astronomiske enheter istedenfor SI enheter, som betyr at vi

- regner tid i år (yr) istedenfor sekunder.
- regner avstander i astronomiske enheter (AU) istedenfor meter[15].
- regner masser i solmasser (SM) istedenfor kilogram.

Dette har effekten at

- gravitasjonskonstanten blir $G = 4\pi^2\,\mathrm{AU}^3\mathrm{yr}^{-2}\mathrm{SM}^{-1}$.
- massen til solen blir $M_{sol} = 1\,\mathrm{SM}$.
- initialposisjonen til jorden blir $x = 1\,\mathrm{AU}$, $y = 0\,\mathrm{AU}$.
- initialhastigheten til jorden blir $v_x = 0\,\mathrm{AU/yr}$, $v_y = 2\pi\,\mathrm{AU/yr}$.

[15] 1 AU er den gjennomsnitlige avstanden mellom jorden og solen.

Bruk `ODESolver` til å løse ligning (10) for jordens bevegelse rundt solen i 10 år, med bruk av initialbetingelsene og parameterene beskrevet over.
For hjelp med å løse andre ordens ODEs, se læreboken side 799-801.

Plott y mot x for de 10 årene, og (forhåpentligvis) se en sirkulær bane i avstand 1 AU.

Hint: Matplotlib holder ikke aksene proporsjonale på plott, som kan få en sirkulær bane til å se epiltisk ut. For å løse dette kan du sette inn linjen `matplotlib.pyplot.axis('equal')`.

Filnavn: `Orbits.py`